Bidirectional Variable-Order de Bruijn Graphs

Belazzougui, Djamal

2018-12

Belazzougui , D , Gagie , T , Mäkinen , V , Previtali , M & Puglisi , S J 2018 , ' Bidirectional Variable-Order de Bruijn Graphs ' , International Journal of Foundations of Computer Science , vol. 29 , no. 8 , pp. 1279-1295 . https://doi.org/10.1142/S0129054118430037

http://hdl.handle.net/10138/322870
https://doi.org/10.1142/S0129054118430037

unspecified
acceptedVersion

International Journal of Foundations of Computer Science

Bidirectional Variable-Order de Bruijn Graphs*

Djamal Belazzougui
CERIST, Algeria
djamal.belazzougui@gmail.com

Travis Gagie
Diego Portales University and CEBIB, Chile
travis.gagie@gmail.com

Veli Mäkinen
Helsinki Institute for Information Technology,
University of Helsinki, Finland
veli.makinen@cs.helsinki.fi

Marco Previtali
University of Milano-Bicocca
marco.previtali@disco.unimib.it

Simon J. Puglisi
Helsinki Institute for Information Technology,
University of Helsinki, Finland
simon.j.puglisi@gmail.com

Received (Day Month Year)
Accepted (Day Month Year)
Communicated by (xxxxxxxxxx)

Compressed suffix trees and bidirectional FM-indexes can store a set of strings and support queries that let us explore the set of substrings they contain, adding and deleting characters on both the left and right, but they can use much more space than a de Bruijn graph for the strings. Bowe et al.'s BWT-based de Bruijn graph representation (Proc. Workshop on Algorithms for Bioinformatics, pp. 225-235, 2012) can be made bidirectional as well, at the cost of increasing its space usage by a small constant, but it fixes the length of the substrings. Boucher et al. (Proc. Data Compression Conference, pp. 383-392, 2015) generalized Bowe et al.'s representation to support queries about variable-length substrings, but at the cost of bidirectionality. In this paper we show how

*A preliminary version of this paper was presented at the 12th Latin American Theoretical Informatics Symposium (LATIN '16). This research was done while the fourth author was visiting the University of Helsinki and the first and second authors were employed there. It was partly funded by Academy of Finland grants 268324, 2845984 and 294143.

to make Boucher et al.'s variable-order implementation of de Bruijn graphs bidirectional.

Keywords: de Bruijn graphs; Burrows-Wheeler Transform; bidirectional FM-index.

1. Introduction

Suppose we have a set of strings and we want to build a compact data structure that supports queries to let a user (a person or a piece of software) explore efficiently the set of substrings those strings contain. Such exploration is useful for, e.g., approximate pattern matching and *de novo* genome assembly. We could build a compressed suffix tree [1, 26, 27] or a bidirectional FM-index [4, 15, 28] and support the following queries:

search(α) returns true and information allowing fast evaluation of other queries if the pattern α occurs in any of the strings, and returns false otherwise;
add-left(c) prepends c to the current pattern and returns true if the resulting pattern occurs in any of the strings, and returns false otherwise;
add-right(c) appends c to the current pattern and returns true if the resulting pattern occurs in any of the strings, and returns false otherwise;
delete-left deletes the first character of the current pattern;
delete-right deletes the last character of the current pattern;
count returns the number of occurrences of the current pattern in the set of strings;
locate returns the positions of the occurrences of the current pattern in the set of strings.

These queries are powerful enough for the user to recover the set of strings, so we cannot hope to compress the structures more than we could *losslessly* compress the strings themselves. It does not help us that the user is interested only in the *set* of substrings rather than their *multiset*. More specifically, compressed suffix trees and bidirectional FM-indexes are mainly based on Burrows-Wheeler Transforms (BWTs) [8] and longest common prefix (LCP) arrays [14] of the strings, whose total length is proportional to that of the strings themselves. Standard unidirectional FM-indexes do not use LCP arrays but they support adding and deleting on only one side and they still need a BWT of all the strings. With an LCP array as well they can support deleting on both sides. For simplicity, in this paper we assume the alphabet size is constant.

If the user is interested only in substrings up to a given maximum length K, however, then we can support less powerful queries that cannot be used to recover the strings, and thus perhaps save more space via *lossy* compression. For example, the strings abracabradabra$ and abradabracadbra$ have the same set of distinct 5-tuples (i.e. substrings of length 5) and so cannot be uniquely reconstructed from that set; for $K \leq 4$, the number of distinct K-tuples in each string is less than its length, decreasing to six (i.e., the characters a, b, c, d and r) for $K = 1$. If we use a hash-based implementation of the strings' Kth-order de Bruijn graph (see, e.g., [5] and references therein) then we can support the following queries while using space

bounded in terms of the number of distinct K-tuples in the strings:

search$_{=K}(\alpha)$ returns true and information allowing fast evaluation of other queries if the pattern α has length K and occurs in any of the strings, and returns false otherwise;

add-left-delete-right(c) prepends c to the current pattern and deletes the last character and returns true if the resulting pattern occurs in any of the strings, and returns false otherwise;

add-right-delete-left(c) appends c to the current pattern and deletes the last character and returns true if the resulting pattern occurs in any of the strings, and returns false otherwise.

Bowe, Onodera, Sadakane and Shibuya's [7] (see also [17]) gave a BWT-based implementation of de Bruijn graphs — which we henceforth refer to as BOSS, for the authors' initials — which supports only one of add-left-delete-right(c) or add-right-delete-left(c) but, as we explain in Section 3, it is easy to add support for both at the cost of increasing the space by a small constant factor. We note that BOSS's BWT is not of the strings but of the edge labels in the de Bruijn graphs, so its length is also bounded in terms of the number of distinct K-tuples.

Boucher, Bowe, Gagie, Puglisi and Sadakane [6] described *variable-order de Bruijn graphs*, a generalization of the BOSS representation, to support search(α) with any pattern α of length *at most* K, and either add-left(c) and delete-right or add-right(c) and delete-left, with the restriction that the pattern always has length at most K. To do this, they added an longest common suffix (LCS) array with the number of entries equal to the length of the BWT of the edge labels and each entry between 0 and K, which increased the space bound by a factor of $\lg K$. As far as we know, similarly generalizing a hash-based implementation of a Kth-order de Bruijn graph would increase the space bound by a factor of K. Apart from the space increase, the main drawback to Boucher et al.'s version of BOSS is that it does not support adding and deleting on both sides, and the easy extension that makes the original BOSS bidirectional does not seem to work here.

In this paper we show how to make Boucher et al.'s variable-order implementation of de Bruijn graphs bidirectional. That is, our version efficiently and simultaneously supports all of the following queries:

search$_{\leq K}(\alpha)$ returns true and information allowing fast evaluation of other queries if the pattern α has length at most K and occurs in any of the strings, and returns false otherwise;

add-left$_{\leq K}(c)$ prepends c to the current pattern and returns true if the resulting pattern has length at most K and occurs in any of the strings, and returns false otherwise;

add-right$_{\leq K}(c)$ appends c to the current pattern and returns true if the resulting pattern has length at most K and occurs in any of the strings, and returns false otherwise;

Table 1. A comparison of (unidirectional) FM-indexes, bidirectional FM-indexes, compressed suffix trees, hash-based de Bruijn graph (dBG) implementations, BOSS, variable-order BOSS and our data structure.

data structure	compression	LCP/LCS needed	pattern length	bidirectional
FM-index [11]	lossless	no	unrestricted	no
bidirectional FM-index [4, 15, 28]	lossless	no	unrestricted	yes
compressed suffix tree [1, 26, 27]	lossless	yes	unrestricted	yes
hash-based dBG (e.g., [5])	lossy	no	exactly K	yes
BOSS [7], Obs. 1	lossy	no	exactly K	yes
variable-order BOSS [6]	lossy	yes	at most K	no
this paper	lossy	yes	at most K	yes

delete-left deletes the first character of the current pattern (as before);
delete-right deletes the last character of the current pattern (as before).

Table 1 compares (unidirectional) FM-indexes, bidirectional FM-indexes, compressed suffix trees, hash-based de Bruijn graph (dBG) implementations, BOSS, variable-order BOSS and our data structure. Our main idea is that, while with a bidirectional FM-index we store a BWT of the set of strings and a BWT of their reverses and keep the intervals for the current pattern synchronized by counting characters, now we store BWTs for the forward and reversed edge labels and keep the intervals for the current pattern synchronized by counting *distinct* characters. In Section 2 we review FM-indexes and bidirectional FM-indexes. In Section 3 we review de Bruijn graphs and BOSS and explain how to make it bidirectional, then review variable-order BOSS. Finally, in Section 4 we explain how to add bidirectionality to variable-order BOSS.

2. FM-Indexes and Bidirectional FM-Indexes

The suffix array [19] of a string $S[1..n]$ is an array $A[1..n]$ containing a permutation of the integers $[1..n]$ such that $S[A[1]..n] < S[A[2]..n] < \cdots < S[A[n]..n]$. In other words, $A[j] = i$ iff $S[i..n]$ is the j^{th} suffix of S in lexicographical order. The Burrows-Wheeler Transform (BWT) of a string $S[1..n]$ is the permutation of S's characters derived from A. In particular, for $2 \leq i \leq n$, if $S[i..n]$ is the lexicographically jth suffix of S, then $\mathsf{BWT}(S)[j] = S[i-1]$; if S itself is lexicographically ranked j among its suffixes, then $\mathsf{BWT}(S)[j] = S[n]$. For convenience, it is often assumed that S ends with a unique delimiter symbol $S[n] = \$$ lexicographically less than any character in the alphabet. For example,

$$\mathsf{BWT}(\mathsf{abracabradabra\$}) = \mathsf{ard\$crraaaaabbb}\,.$$

The BWT was first proposed as a pre-processing step for data compression [8], because it gathers together characters that precede similar contexts, but perhaps its most important use has come since, in the design of indexing data structures.

a	$abracabradabr	a	$abracabradabr
r	a$abracabradab	r	a$abracabradab
d	abra$abracabra	d	abra$abracabra
$	abracabradabra	$	abracabradabra
c	abradabra$abra	c	abradabra$abra
r	acabradabra$ab	r	acabradabra$ab
r	adabra$abracab	r	adabra$abracab
a	bra$abracabrad	a	bra$abracabrad
a	bracabradabra$	a	bracabradabra$
a	bradabra$abrac	a	bradabra$abrac
a	cabradabra$abr	a	cabradabra$abr
a	dabra$abracabr	a	dabra$abracabr
b	ra$abracabrada	b	ra$abracabrada
b	racabradabra$a	b	racabradabra$a
b	radabra$abraca	b	radabra$abraca

Fig. 1. To prepend an r to the pattern a with an FM-index for abracabradabra$, we find the ranks of the occurrences of r in the range of the BWT containing characters preceding occurrences of a in the string (shown in red on the left), then compute the range those r's are mapped to by stable sorting (shown in red on the right).

Ferragina and Manzini [11] noted that, if we have pre-computed the number of characters in $\mathsf{BWT}(S)$ (or, equivalently, S) less than each distinct character, have a data structure supporting fast rank queries over $\mathsf{BWT}(S)$ — i.e., when given a character c and a position i, it quickly returns how many occurrences of c there are in the prefix $\mathsf{BWT}(S)[1..i]$ — and know the interval in $\mathsf{BWT}(S)$ containing characters preceding a pattern P then, for any character c we can quickly compute the interval in $\mathsf{BWT}(S)$ containing characters preceding occurrences of cP. Specifically, if query $\mathsf{rank}_c(i)$ returns the number of occurrences of the symbol c up to position i in a string, by the definition of the BWT, if the interval for P is $\mathsf{BWT}(S)[i..j]$ then the interval for cP is

$$\mathsf{BWT}(S)\left[|\{x : x \prec c\}| + \mathsf{rank}_c(i-1) + 1..|\{x : x \prec c\}| + \mathsf{rank}_c(j)\right] .$$

For example, as shown in Figure 1, if $S = \mathsf{abracabradabra\$}$ then the interval for a is $\mathsf{BWT}(S)[2..7]$, $|\{x : x \prec r\}| = 12$, $\mathsf{rank}_r(2) = 0$, $\mathsf{rank}_r(7) = 3$ and the interval for ra is $\mathsf{BWT}(S)[13..15]$.

The number of occurrences of a pattern in S is just the length of its interval in $\mathsf{BWT}(S)$ and if we keep track of the intervals (using, say, a stack) as we add characters to the left of a pattern, we can always delete those characters again, so with a basic FM-index we can support the following queries:

search(α) returns true and information allowing fast evaluation of other queries if the pattern α occurs in any of the strings, and returns false otherwise;

a	$abracabradabr	0
r	a$abracabradab	0
d	abra$abracabra	1
$	abracabradabra	4
c	abradabra$abra	4
r	acabradabra$ab	1
r	adabra$abracab	1
a	bra$abracabrad	0
a	bracabradabra$	3
a	bradabra$abrac	3
a	cabradabra$abr	0
a	dabra$abracabr	0
b	ra$abracabrada	0
b	racabradabra$a	2
b	radabra$abraca	2

Fig. 2. To delete the last three characters of the pattern abra, we find the positions of the last and first LCP values less than 1 before and after the interval for abra, respectively, which tell us the beginnings of the intervals for a and for the lexicographically next character b.

add-left(c) prepends c to the current pattern and returns true if the resulting pattern occurs in any of the strings, and returns false otherwise;
delete-left deletes the first character of the current pattern;
count returns the number of occurrences of the current pattern in the set of strings.

More interestingly, if we can perform fast select queries over $\mathsf{BWT}(S)$ and we store as well a sampled suffix array (composed only of every xth suffix, see e.g., [20]) and a wavelet tree [13] (a data structure supporting various types of range queries [12]) over a longest common prefix (LCP) array for S, in which $\mathsf{LCP}[1] = 0$ and $\mathsf{LCP}[i]$ is the length of the longest common prefix between the lexicographically $(i-1)$st and ith suffixes of S, then we can support also the following queries:

delete-right deletes the last character of the current pattern;
locate returns the positions of the occurrences of the current pattern in the set of strings.

For example, Figure 2 shows how, if we have the the BWT and a wavelet tree for the LCP array of $S =$ abracabradabra$ and know the interval for abra is $\mathsf{BWT}(S)[3..5]$, then we can delete the three last characters of the pattern and find the interval for a by querying the wavelet tree to find the positions of the last value less than 1 in $\mathsf{LCP}[1..2]$ and the first first value less than 1 in $\mathsf{LCP}[6..15]$. We refer the reader to Navarro's recent text [21] for details on these auxiliary data structures and how they are used to support these queries.

Lam et al. [16] (see also, e.g., [4, 15, 28]) showed how to make an FM-index

bidirectional, allowing us to add (but not delete) characters on both the left and right. To do this, we store BWTs both for S and for its reverse S^R. If we add a character on the left of the pattern, we update the interval in $\mathsf{BWT}(S)$ as normal. To update the interval in $\mathsf{BWT}(S^R)$, we use the fact that it will be a sub-interval of the previous interval. For example, as shown in Figure 3, if we add an r on the right of a then $\mathsf{BWT}(S^R)$, shown on the right, the interval narrows from $\mathsf{BWT}(S^R)[2..7]$ to $\mathsf{BWT}(S^R)[5..7]$. Specifically, if the previous pattern is preceded in S by ℓ characters strictly less than the character we added to the left and by e copies of that character — which we can determine by examining the contents of the interval for the previous pattern in $\mathsf{BWT}(S)$ — then the sub-interval for the new pattern will start ℓ positions into the interval for the previous pattern in $\mathsf{BWT}(S^R)$ and have length e. In our example, $\ell = e = 3$ since a is preceded by three characters ($, c, d)) less than r and three occurrences of r in S. Adding a character on the right of the pattern is symmetric, with the roles of the BWTs reversed, so we now have support for the last query:

add-right(c) appends c to the current pattern and returns true if the resulting pattern occurs in any of the strings, and returns false otherwise.

In summary, a bidirectional FM-index for a string or set of strings uses BWTs whose total length is proportional to the total length of the strings, and supports adding characters on both the left and right.

3. BOSS and Variable-Order BOSS

In bioinformatics, a Kth-order de Bruijn graph for a string or set of strings is a directed graph in which there is a node for each distinct K-tuple in the strings and an edge from a node u to a node v if there is a $(K+1)$-tuple whose first and last K characters are u and v, respectively. If there is such an edge, it is said to be labelled with the last character of v. Pevzner et al. [25] introduced this definition — different from that in combinatorics [10] — as a tool for *de novo* genome assembly.

For small K, the Kth-order de Bruijn graph can contain much less information than the strings. For example, as we noted in Section 1, the strings abracabradabra$ and abradabracadbra$ have the same set of distinct 5-tuples and so cannot be uniquely reconstructed from that set. To see why this kind of information loss can actually be useful when trying to assemble a genome, suppose we are trying to reconstruct the unknown string abracabradabra$ from the substrings abracabr, bracabra, racabrad, acabrada, cabradab, abradabr, bradabra and radabra$. The total length of these substrings is 64 and so, if we build an FM-index for their concatenation with separator characters or a multi-string FM-index [3] for them, to use in the assembly, the underlying BWT will have far more characters than abracabradabra$ itself. This is partly because, even if we were given abracabradabra$, to recover the substrings we would still need to know that there are eight of them and where they start and end; in some sense, this information is encoded in the extra

a	$abracabradabr	bracabradabra$	a
r	a$abracabradab	racabradabra$a	b
d	abra$abracabra	radabra$abraca	b
$	abracabradabra	ra$abracabrada	b
c	abradabra$abra	abradabra$abra	c
r	acabradabra$ab	abra$abracabra	d
r	adabra$abracab	abracabradabra	$
a	bra$abracabrad	acabradabra$ar	r
a	bracabradabra$	adabra$abracar	r
a	bradabra$abrac	a$abracabradar	r
a	cabradabra$abr	bradabra$abrac	a
a	dabra$abracabr	bra$abracabrad	a
b	ra$abracabrada	cabradabra$abr	a
b	racabradabra$a	dabra$abracabr	a
b	radabra$abraca	$abracabradabr	a

a	$abracabradabr	bracabradabra$	a
r	a$abracabradab	racabradabra$a	b
d	abra$abracabra	radabra$abraca	b
$	abracabradabra	ra$abracabrada	b
c	abradabra$abra	abradabra$abra	c
r	acabradabra$ab	abra$abracabra	d
r	adabra$abracab	abracabradabra	$
a	bra$abracabrad	acabradabra$ab	r
a	bracabradabra$	adabra$abracab	r
a	bradabra$abrac	a$abracabradab	r
a	cabradabra$abr	bradabra$abrac	a
a	dabra$abracabr	bra$abracabrad	a
b	ra$abracabrada	cabradabra$abr	a
b	racabradabra$a	dabra$abracabr	a
b	radabra$abraca	$abracabradabr	a

Fig. 3. To prepend an r to the pattern a with a bidirectional FM-index for abracabradabra$, we update the range in our forward BWT (**left**) as with a standard unidirectional BWT. To update the range in our reverse BWT (**right**), we use the forward BWT to compute 1) the number ℓ of characters lexicographically less than r that precede occurrences of a in the string, in this case 3 ($, c, d); 2) the number e of occurrences of r that precede occurrences of a in the string, in this case also 3. The range for ra in the reverse BWT starts ℓ positions into the range for a and has length e. To delete the r again we use select queries over relevant LCP arrays (not shown).

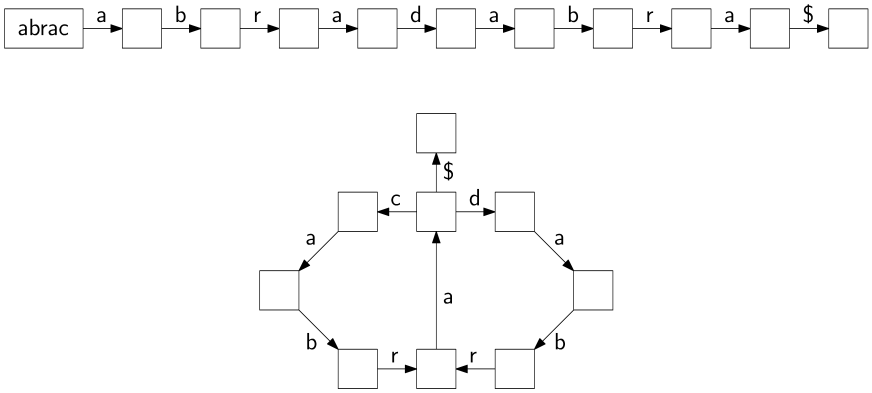

Fig. 4. We can recover the string abracabradabra$ from the the 5th-order de Bruijn graph **(top)** for the substrings abracabr, bracabra, racabrad, acabrada, cabradab, abradabr, bradabra and radabra$, but not from the smaller 3rd-order de Bruijn graph **(bottom)**.

characters in the BWT.

Overlaps between the substrings tend to cause runs of the same character in the BWT and we can compress these easily, but the the LCP array and suffix array sample also grow and are harder to compress. Worse, the values in those arrays can be up to about the lengths of the overlaps and the total length of the substrings, respectively. Therefore, it can be important to discard information about the substrings while trying to retain information about the underlying string they came from. Turning the substrings into the 5th-order de Bruijn graph shown at the top of Figure 4 (whose nodes correspond to 5-tuples but whose edges correspond to 6-tuples in the substrings) does that: it contains enough information to assemble abracabradabra$, because it is a path, but not enough to recover the substrings.

The 3rd-order de Bruijn graph for the substrings is even smaller, as shown at the bottom of Figure 4, although we cannot reconstruct abracabradabra$ from it. In general, however, we may not be able to reconstruct the underlying string even from the substrings themselves, and reducing the space used is important enough to sacrifice some information, at least in the early stages of the assembly. Of course, even though this graph looks small, since a pointer is so much larger than a character, its space usage depends on how we implement it. For example, as stated in Section 1, if we use a hash-based implementation then we can support bidirectional navigation in the graph (i.e., appending and simultaneously deleting on both the left and right of the current pattern, which correspond to crossing edges in the graph forwards and backwards), with K fixed. In this paper, however, we are more interested in Bowe et al.'s BOSS representation and how it can be extended.

To build the BOSS representation of a de Bruijn, we create dummy nodes by padding normal nodes' prefixes on the left with copies of the delimiter symbol $ until every normal node has a path of length at least K leading to it. We then sort

the nodes of the graph into right-to-left lexicographic order and write for each node the labels of the edges leaving it in lexicographic order to obtain the *edge-BWT* (so-called, because it is composed of edge labels). We also store the in- and out-degrees of the nodes as bitvectors. Figure 5 shows the edge-BWTs abbbcd$rrraaaaa and bb$cdrraaa of the 5th- and 3rd order de Bruijn graphs for the substrings in our example, in the center and on the right, respectively, together with some auxiliary arrays on the left that we will explain later. Since we assume the alphabet size is constant, BOSS uses a constant number of bits per edge (asymptotically approximately 4 in the case of DNA sequences).

The key observation behind BOSS is that, considering the nodes in right-to-left lexicographic order, edges with lexicographically smaller labels go to earlier nodes; for edges with the same label, the order of their destination nodes is the same as the order of their source nodes (although two edges with the same label are allowed to point to the same node) so, for any character c, the rth outgoing edge labelled $c \neq \$$ is also the rth incoming edge labelled c. Figure 6 illustrates this for the 3rd-order de Bruijn graph of our example, except that it does not include the edge labelled $ or the node ra$ for simplicity and because, for many applications, we do not want to cross edges labelled with delimiters. Because of this property, if we already know the interval in the edge-BWT containing the labels of the edges leaving a node u and we want to follow the edge e labelled $c \neq \$$ leaving u — i.e., the add-right-delete-left(c) query described in Section 1 — then we can compute the interval for the node v reached by e: we find the rank r of the occurrence of c in u's interval; find the number ℓ of edges with labels strictly less than c (ignoring $s); find the first partial sum at least $\ell + r$ of the nodes' in-degrees, which is v's index; and use the bitvector encoding the nodes' out-degrees to find v's interval.

For example, if we know the interval for dab in the edge-BWT for the 3rd-order de Bruijn graph for our example is [7] and we want to follow the edge labelled r to abr, then we compute $r = \mathsf{rank}_{\mathsf{r}}(7) = 2$; find the first partial sum of the in-degrees that is at least 9 (2 plus the the number 7 of edges with labels less than r but not $), which is the 8th partial sum; and use the bitvector encoding the out-degrees to find the single-entry interval [10].

Bowe et al. noted that, if we have the interval for v, we can find the intervals of its predecessors using something like the reverse procedure: from v's index we compute the range of its incoming edges; for each such edge, we use a select query to find its label in the edge-BWT; then we again use the bitvector for the out-degrees to find the interval of labels on edges leaving the same node. This is not symmetric to add-right-delete-left, however, since we do not learn the first characters of v's predecessors. We can determine each first character by walking backwards across $K - 1$ edges (which is always possible because of the dummy nodes we add when building BOSS), but of course this incurs an $\Omega(K)$-factor increase in the query time, so it is unclear whether this approach merits being called bidirectionality.

For example, if we know the interval for abr is [10] and we want to find the intervals of its predecessors, then we look at its in-degree and see that its in-edges

0		0	$$$$$	a	1
0	$	1	$$$$a	b	1
1	a	1	braca	b	1
1	a	1	brada	b	1
1	$	1	$abra	c	1
4	a	1	cabra	d	1
4	a	1	dabra	$	1
0	$	1	$$$ab	r	1
2	b	1	racab	r	1
2	b	1	radab	r	1
0	$	1	abrac	a	1
0	c	1	abrad	a	1
0	$	1	$$abr	a	1
3	r	1	acabr	a	1
3	r	1	adabr	a	1

1	aca	b	1
1	ada	b	1
1	bra	$	0
	bra	c	0
	bra	d	1
1	cab	r	1
1	dab	r	1
1	rac	a	1
1	rad	a	1
2	abr	a	1

Fig. 5. The BOSS representation for the 5th-order **(center)** and 3rd-order **(right)** de Bruijn graphs of abracabr, bracabra, racabrad, acabrada, cabradab, abradabr, bradabra and radabra$, with the in-degrees of the nodes (shown as numbers, not bitvectors), the nodes themselves, the edge-BWTs, and bitvectors indicating whether each edge label is the last for its node (and thus encoding the the node's out-degree). The auxiliary arrays **(left)** indicate the lengths of the longest common suffixes of the nodes in the 5th-order graph and the labels their incoming edges would have if their directions were reversed.

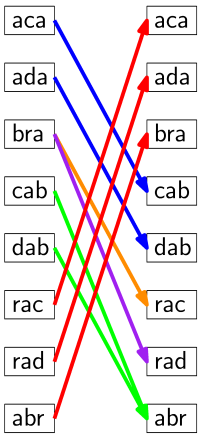

Fig. 6. The 3rd-order de Bruijn graphs of abracabr, bracabra, racabrad, acabrada, cabradab, abradabr, bradabra and radabra$ depicted as a bipartite graph, with edges' colours indicating their labels: red for a, blue for b, orange for c, purple for *d* and green for r. Edges of the same colour can merge but not cross. We do not include the edge labelled $ or the node ra$ for simplicity and because, for many applications, we do not want to cross edges labelled with delimiters.

are the first and second (and only) ones labelled r; we find the first and second occurrences of r in the edge-BWT, which happen to be consecutive, and see from the bitvector for the out-degrees that the intervals for the predecessors are [6] and [7].

Of course, if we simply store the first character of each of v's predecessors, in the right-to-left lexicographic order of the predecessors, then we achieve bidirectionality at the cost of approximately doubling the BOSS's space. The second column of Figure 5 shows the first characters of the nodes' predecessors in the 5th-order graph for our example.

Observation 1. *We can make BOSS bidirectional, such that traversing an edge forwards or backwards takes constant time when the alphabet size is constant, at the cost of approximately doubling the space required.*

We have not discussed how to choose the order K of the de Bruijn graph for assembling a genome. This is an important concern because if K is too small then the graph becomes dense and uninformative, but if it is too large then the graph fragments into many disconnected pieces. There is software available that recommends a good value of K for a set of substrings (see, e.g., [9]) but sometimes no single K works well. This has become a particular concern with the development with single-cell DNA sequencing, which can generate read sets with extremely variable coverage. Some assemblers, such as IDBA [22, 23, 24] and SPAdes [2], try several increasing orders, adding the results from each run to the input data. This slows the assembly down, however, and to remain practical these assemblers must skip most orders. Lin and Pevzner [18] proposed manifold de Bruijn graphs, in which the nodes are a suffix-free set of variable-length strings with the lengths chosen to improve assembly, but without offering a practical implementation.

Boucher et al. [6] noted that if we add a longest common suffix (LCS) array to BOSS — which stores the length of the longest common suffix between the source node for each edge and that of the preceding edge, and thus increases the space bound by an $\mathcal{O}(\log K)$-factor — then we can vary the order of the graph on the fly, up to the value of K chosen for the basic BOSS. This means, among other things, that their variable-order version of BOSS can be used as a practical implementation of manifold de Bruijn graphs. For example, the leftmost column of Figure 5 shows the LCS array for the 5th-order de Bruijn graph for our example.

To see how this works, consider the BOSS representations for the 5th- and 3rd-order graphs in Figure 5. By the definition of a de Bruijn graph, each node v in the 3rd-order graph is a suffix of at least one node in the 5th-order graph and, if we take the union of the sets of edges leaving nodes suffixed by v in the 5th-order graph, we obtain the set of edges leaving v. This means we can use the intervals of nodes that share suffixes of length $k \leq K$ as if they were the intervals for nodes in a kth-order de Bruijn graph. We can decrease the current order k, potentially widening the current interval, by extending the interval to include the first and last nodes in the

Kth-order de Bruijn graph that share a suffix of the new desired length with the first and last node in the current interval. If we want to cross an edge labelled c in the kth-order graph, we can choose any occurrence of c in the current interval and cross the corresponding edge in the Kth-order graph, then use the LCS array to widen the resulting interval to correspond to a node in the kth-order graph.

For example, if we know the union [13..15] of the intervals for nodes suffixed by abr in the 5th-order graph and want to find the union for bra, then we can choose any a in the former interval in the edge-BWT, say the one labelling the edge leaving acabr, and follow it in the 5th-order graph. We arrive at cabra and, using the LCS array to find the union of the intervals of nodes suffixed by bra, we obtain [5..7].

4. Adding Bidirectionality

As mentioned in Section 1, our main idea is to store BWTs for the forward and reversed edge labels and keep the intervals for the current pattern synchronized by counting *distinct* characters. Before we explain this idea, however, we feel it is instructive to review why the two most obvious approaches — i.e., straightforward generalizations to Boucher et al.'s data structure of the approaches to bidirectionality explained in Sections 2 and 3 — apparently fail.

Lam et al.'s [16] and other bidirectional FM-indexes use the fact that if we form a matrix by taking the cyclic shifts of S and put them in right-to-left lexicographic order — as for computing $\mathsf{BWT}(S^R)$ — then the characters in the interval for P in $\mathsf{BWT}(S)$ appear in the matrix in the $(|P|+1)$st column from the right in the interval for P^R in $\mathsf{BWT}(S^R)$ in lexicographic order and with the same frequencies; the same is true if we swap the roles of S and S^R and P and P^R and form the matrix with normal lexicographic sorting. For example, in Figure 3, the characters in the interval for a in $\mathsf{BWT}(S)$ — i.e., r, d, \$, c, r, r — shown in the top-left matrix, appear in lexicographic order immediately to the left of the occurrences of a shown in red in the top-right matrix. This symmetry does not hold for BOSS: to see why not, consider that bra is preceded only by a but followed by \$, c and d in $S =$ abracabradabra\$; therefore, the interval for bra in the BOSS representation of a 3rd-order de Bruijn graph has length 3, but the interval for rba has length only 1 in the BOSS representation of a 3rd-order de Bruijn graph for S^R.

Just as the second column in Figure 5 — i.e., the predecessors' first characters in the 5th-order graph — would be the leftmost row of the matrix for the 6th-order graph, if we could compute efficiently any entry of the grey matrix for the 5th-order graph, then we could use the ideas behind Observation 1 to obtain a bidirectional variable-order BOSS. For example, if we know the interval for bra is [5..7] then we use Bowe et al.'s technique and the LCS array to determine there is a single predecessor of bra and that its interval is [13..15]; if we could compute efficiently somehow that in the second column of the matrix there are copies of a in the interval for bra, then we would know that bra's predecessor is abr. We see no way of supporting efficient access to the matrix directly, however. Of course, similarly to

how a bidirectional FM-index for $S = \mathsf{abracabradabra\$}$ implicitly supports access to the matrices conceptually used to compute $\mathsf{BWT}(S)$ and $\mathsf{BWT}(S^R)$, the technique we describe below supports computation of the edge-BWTs.

Given a string or set of strings, we pad them on the left and the right with copies of a delimiter symbol \$ such that for every node v in the $(K+1)$st-order de Bruijn graph, there is a path from $\$^{K+1}$ to v and vice versa. We build Boucher et al.'s variable-order BOSS representations for the resulting set of strings, and for the resulting set of strings reversed. Consider any k-tuple α with $k \leq K$, let V_L be the set of nodes prefixed by α, let V_R be the set of nodes suffixed by α, let U be the set of predecessors of nodes in V_L and let W be the set of successors of nodes in V_R.

Lemma 2. *The same set of characters appear as the last characters of nodes in W as appear as the $(k+1)$st characters of nodes in V_L, and the same set of characters appear as the first characters of nodes in U as appear as the $(k+1)$st characters from the right of nodes in V_R.*

Proof. By the definition of a de Bruijn graph, the set of last characters of nodes in W are those that appear immediately to the right of occurrences of α in the padded strings (possibly not with the same frequencies). That is, a character c is in that set if and only if αc occurs as a $(k+1)$-tuple, which is equivalent to αc being the prefix of some node (which is also prefixed by α).

Symmetrically, the set of first characters of nodes in U are those immediately to the left of occurrences of α in the padded strings (possibly not with the same frequencies). That is, a character c is in that set if and only if $c\alpha$ occurs as a $(k+1)$-tuple, which is equivalent to $c\alpha$ being the suffix of some node (which is also suffixed by α). □

Lemma 2 means that we have at least a weaker form of the symmetry behind bidirectional FM-indexes: we generally cannot tell the frequency with which a particular character occurs in the nodes to the right or left of a prefix or suffix α by looking at the intervals for α and α^R in the edge-BWTs, but we can at least tell which characters occur like that. In fact, with the LCP and LCS arrays for the nodes, this weak form of symmetry is sufficient for bidirectionality: if we already know the intervals for α and α^R in the edge-BWTs for the padded strings and their reverses, then we can find the interval for αc in the edge-BWT for the padded strings as normal; to find the interval for $(\alpha c)^R$ in the edge-BWT of the reverses, we count the number d of distinct characters there are strictly smaller than c in the interval for α in the edge-BWT of the padded strings, and use the LCP array for the reverses to skip over the intervals for that many nodes suffixed by α^R in the edge-BWT of the reversed strings.

This procedure is essentially the same as Lam et al.'s but, since we cannot rely on the counts of characters being the same, we use the counts of distinct characters. Notice this approach also works for bidirectional FM-indexes: in the example shown

in Figure 3, we can find the interval for ra in the reverse BWT by skipping over the intervals for $a, ca and da. A disadvantage to counting distinct characters instead of counting characters is that, since adding a character to the left or right now requires queries on the LCP arrays, all queries take $\mathcal{O}(\log K)$ time using wavelet trees over those arrays.

Since we are storing the LCP arrays, which take $\mathcal{O}(\log K)$ bits per entry, we can relax our assumption that the alphabet size is constant: we still use $\mathcal{O}(\log K)$ bits per edge as long as the alphabet size is polynomial in K. We can perform add-right$_{\leq K}$ or, symmetrically, add-left$_{\leq K}$ in $\mathcal{O}(\log K)$ time. By repeating these queries at most K times we can perform search$_{\leq K}$ in $\mathcal{O}(K \log K)$ time. To perform delete-left we update the interval in the edge-BWT for the padded edges using its LCP array, and we update the interval in the edge-BWT for the reverses using a select query and its LCP. Performing delete-right is symmetric. Summarizing, we have the following theorem:

Theorem 3. *Given K and a string or set of strings over a alphabet of size at most polynomial in K, we can store a data structure in $\mathcal{O}(\log K)$ bits per edge in the $(K+1)$st-order de Bruijn graph for the strings (padded on the left and right such that there is a path from each node to $\$^{K+1}$ and vice versa), that supports the following operations:*

search$_{\leq K}(\alpha)$ *returns* true *and information allowing fast evaluation of other queries if the pattern α has length at most K and occurs in any of the strings, and returns* false *otherwise;*

add-left$_{\leq K}(c)$ *prepends c to the current pattern and returns* true *if the resulting pattern has length at most K and occurs in any of the strings, and returns* false *otherwise;*

add-right$_{\leq K}(c)$ *appends c to the current pattern and returns* true *if the resulting pattern has length at most K and occurs in any of the strings, and returns* false *otherwise;*

delete-left *deletes the first character of the current pattern;*

delete-right *deletes the last character of the current pattern.*

The query search$_{\leq K}$ *takes $\mathcal{O}(K \log K)$ time and the other queries take $\mathcal{O}(\log K)$ time.*

5. Conclusion and Future Work

We have shown how to make the variable-order de Bruijn graph representation of Boucher et al.'s bidirectional without asymptotically increasing space usage. Implementing this proposal in order to guage its effectiveness in practice, is perhaps the most pressing direction for future work. It may also be possible to apply the ideas behind Theorem 3 to speed up forward searching in Sirén's GCSA2 index [29] for genomic reference sequences, perhaps at the cost of doubling the space used.

References

[1] A. Abeliuk, R. Cánovas and G. Navarro, Practical compressed suffix trees, *Algorithms* **6**(2) (2013) 319–351.

[2] A. Bankevich, S. Nurk, D. Antipov, A. A. Gurevich, M. Dvorkin, A. S. Kulikov, V. M. Lesin, S. I. Nikolenko, S. Pham, A. D. Prjibelski *et al.*, SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing, *Journal of computational biology* **19**(5) (2012) 455–477.

[3] M. J. Bauer, A. J. Cox and G. Rosone, Lightweight algorithms for constructing and inverting the BWT of string collections, *Theoretical Computer Science* **483** (2013) 134–148.

[4] D. Belazzougui, F. Cunial, J. Kärkkäinen and V. Mäkinen, Versatile succinct representations of the bidirectional Burrows-Wheeler transform, *European Symposium on Algorithms*, Springer (2013), pp. 133–144.

[5] D. Belazzougui, T. Gagie, V. Mäkinen and M. Previtali, Fully dynamic de Bruijn graphs, *International Symposium on String Processing and Information Retrieval*, Springer (2016), pp. 145–152.

[6] C. Boucher, A. Bowe, T. Gagie, S. J. Puglisi and K. Sadakane, Variable-order de Bruijn graphs, *Data Compression Conference (DCC), 2015*, IEEE (2015), pp. 383–392.

[7] A. Bowe, T. Onodera, K. Sadakane and T. Shibuya, Succinct de Bruijn graphs, *International Workshop on Algorithms in Bioinformatics*, Springer (2012), pp. 225–235.

[8] M. Burrows and D. J. Wheeler, A block-sorting lossless data compression algorithm, Tech. Rep. 24, Digital Equipment Corporation (1994).

[9] R. Chikhi and P. Medvedev, Informed and automated k-mer size selection for genome assembly, *Bioinformatics* (2013) p. btt310.

[10] N. G. de Bruijn, A combinatorial problem, *Koninklijke Nederlandse Akademie van Wetenschappen* **49** (1946) 758–764.

[11] P. Ferragina and G. Manzini, Indexing compressed text, *Journal of the ACM (JACM)* **52**(4) (2005) 552–581.

[12] T. Gagie, G. Navarro and S. J. Puglisi, New algorithms on wavelet trees and applications to information retrieval, *Theor. Comput. Sci.* **426** (2012) 25–41.

[13] R. Grossi, A. Gupta and J. S. Vitter, High-order entropy-compressed text indexes, *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, (ACM/SIAM, 2003), pp. 841–850.

[14] J. Kärkkäinen, G. Manzini and S. J. Puglisi, Permuted longest-common-prefix array, *Proc. 20th Annual Symposium on Combinatorial Pattern Matching*, *Lecture Notes in Computer Science* **5577**, (Springer, 2009), pp. 181–192.

[15] G. Kucherov, K. Salikhov and D. Tsur, Approximate string matching using a bidirectional index, *Theoretical Computer Science* **638** (2016) 145–158.

[16] T. W. Lam, R. Li, A. Tam, S. Wong, E. Wu and S.-M. Yiu, High throughput short read alignment via bi-directional BWT, *Bioinformatics and Biomedicine, 2009. BIBM'09. IEEE International Conference on*, IEEE (2009), pp. 31–36.

[17] D. Li, C.-M. Liu, R. Luo, K. Sadakane and T.-W. Lam, MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph, *Bioinformatics* (2015) p. btv033.

[18] Y. Lin and P. A. Pevzner, Manifold de Bruijn graphs, *International Workshop on Algorithms in Bioinformatics*, Springer (2014), pp. 296–310.

[19] U. Manber and G. W. Myers, Suffix arrays: a new method for on-line string searches, *SIAM J. Comp.* **22**(5) (1993) 935–948.

[20] G. Navarro and V. Mäkinen, Compressed full-text indexes, *ACM Computing Surveys* **39**(1) (2007) p. article 2.
[21] G. Navarro, *Compact Data Structures: A Practical Approach* (Cambridge University Press, 2016).
[22] Y. Peng, H. C. Leung, S.-M. Yiu and F. Y. Chin, IDBA–a practical iterative de Bruijn graph de novo assembler, *Annual International Conference on Research in Computational Molecular Biology*, Springer (2010), pp. 426–440.
[23] Y. Peng, H. C. Leung, S.-M. Yiu and F. Y. Chin, Meta-IDBA: a de novo assembler for metagenomic data, *Bioinformatics* **27**(13) (2011) i94–i101.
[24] Y. Peng, H. C. Leung, S.-M. Yiu and F. Y. Chin, IDBA-UD: a de novo assembler for single-cell and metagenomic sequencing data with highly uneven depth, *Bioinformatics* **28**(11) (2012) 1420–1428.
[25] P. A. Pevzner, H. Tang and M. S. Waterman, An Eulerian path approach to DNA fragment assembly, *Proceedings of the National Academy of Sciences* **98**(17) (2001) 9748–9753.
[26] L. Russo, G. Navarro and A. L. Oliveira, Fully compressed suffix trees, *ACM transactions on algorithms (TALG)* **7**(4) (2011) p. 53.
[27] K. Sadakane, Compressed suffix trees with full functionality, *Theory of Computing Systems* **41**(4) (2007) 589–607.
[28] T. Schnattinger, E. Ohlebusch and S. Gog, Bidirectional search in a string with wavelet trees and bidirectional matching statistics, *Information and Computation* **213** (2012) 13–22.
[29] J. Sirén, Indexing variation graphs, *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, SIAM (2017), pp. 13–27.